ValuePack:Value-Based Scheduling Framework for CPU-GPU Clusters

Vignesh T. Ravi[1], Michela Becchi[2], Gagan Agrawal[1], and Srimat Chakradhar[3]

[1] *Dept. of Computer Science and Engg., The Ohio State University Columbus, Ohio*
{raviv, agrawal}@cse.ohio-state.edu

[2] *Dept. of Electrical and Computer Engg., University of Missouri Columbia, Missouri*
{becchim}@missouri.edu

[3] *NEC Laboratories America, Princeton, New Jersey*
{chak}@nec-labs.com

***Abstract*—Heterogeneous computing nodes are becoming commonplace today, and recent trends strongly indicate that clusters, supercomputers, and cloud environments will increasingly host more heterogeneous resources, with some being massively parallel (e.g., GPU). With such heterogeneous environments becoming common, it is important to revisit scheduling problems for clusters and cloud environments.**

In this paper, we formulate and address the problem of *value-driven* scheduling of independent jobs on heterogeneous clusters, which captures both the *urgency* and *relative priority* of jobs. Our overall scheduling goal is to maximize the aggregate value or yield of all jobs. Exploiting the portability available from the underlying programming model, we propose four novel scheduling schemes that can *automatically* and *dynamically* map jobs onto heterogeneous resources. Additionally, to improve the utilization of massively parallel resources, we also propose heuristics to automatically decide when and which jobs can share a single resource.

I. Introduction

In the last few years, multi-core and many-core processor architectures have both become pervasive. The evolution of GPUs along with the introduction of CUDA, and subsequently OpenCL has enabled wide-spread general-purpose computing on specialized graphics processors. Today it is extremely common for desktops, laptops, clusters, and supercomputers to include both multi-core CPUs and GPUs. For instance, 3 out of the top 5 supercomputers (as of November 2011) [9] use both CPUs and Nvidia GPUs. Continuing this trend, cloud environments have also started to embrace the presence of heterogeneous resources. For example, Amazon, a successful cloud provider, and Nimbix [1] introduced GPU instances in 2010 [1] to attract more HPC users. While CPU-GPU clusters are the most popular heterogeneous systems today, other processor architectures are also emerging. Intel has recently announced its Many Integrated Core (MIC) architecture [2], which is likely to be released in late 2012. Accelerated Processing Units (APUs) that integrate CPU and GPU cores into a single die represent yet another flavor of heterogeneous processors. The AMD Fusion and the Intel Sandy Bridge are two popular instances of APUs. As an example of how heterogeneity will feature in future high-end systems, consider the recently announced Stampede system [8], which will have peak performance of more than 10 petaflops. As per the current plan, the Stampede system will include Intel Sandy Bridge processors, with Intel MIC co-processors, and a number of NVIDIA GPUs. Overall, it is clear that clusters, supercomputers, as well as the cloud environments targeting HPC workloads are increasingly involving more heterogeneous resources.

Even though supercomputers are often rated by the peak performance achieved on a single application, in practice they are concurrently used by a number of independent jobs. In such cases, a critical component of the software stack is a *job scheduler*, which aims at improving the resource utilization and maximizing the aggregate throughput. Most large supercomputing centers allocate a given number of *units* to each user or research group; these units are consumed as the users' jobs are executed. Cloud environments, on the other hand, have a notion of *pay-as-you-go* model, where resources are viewed as services. Apart from global throughput, profitability of resource provider, user satisfaction, and/or the priority across users can be important considerations for both supercomputing centers and cloud environments. Job scheduling in supercomputers has been extensively studied over the years [21], [34], [39], [40], [44], [46]. However, we believe that it is compelling to revisit scheduling problems in the context of today's modern heterogeneous clusters (including multi-core CPUs, many-core GPUs, and others), while factoring user satisfaction, profitability of the service providers, and priority across different user types.

TORQUE [10] is an open-source batch scheduler for today's heterogeneous (CPU-GPU) clusters, and is widely used in hundreds of supercomputer centers. We claim that it lacks three important features that are highly desirable today:
(i) First, scheduling schemes within TORQUE are designed with the goal to improve system-wide metrics like the global throughput. In an emerging market-based service world, the users and jobs are becoming more classified. For instance, one user could be more valuable than the other, and one job could be more urgent than the other. We believe that scheduling schemes that can maximize the service providers' profit and/or improve user satisfaction are useful.
(ii) Next, the type of resource (CPU or GPU) to execute a job is currently decided by the user (as configured through the *qsub* command). While this was a reasonable model when there was no portability across CPUs and GPUs, emergence of OpenCL [4] and compilers like PGI CUDA-x86 [6] are now enabling portability across different devices. Such portability is not exploited by TORQUE. Particularly, *when contention on one resource (say GPU) can be much higher than on the other (say multi-core CPU), the uneven resource demands can adversely affect utilization, and user response times*. Scheduling schemes that can *automatically* assign either CPU or GPU to the jobs, exploiting the portability of applications, and taking system load into account, are clearly desirable.

[1]Please see:http://www.nimbix.net/

SC12, November 10-16, 2012, Salt Lake City, Utah, USA
978-1-4673-0806-9/12/$31.00 ©2012 IEEE

(iii) Finally, today GPUs are massively parallel (about 512 cores), and running only one application at a time can sometimes under-utilize the GPU. Hence, it is often preferable to allow GPU sharing among multiple jobs. Currently, TORQUE neither has the mechanism nor the schemes to *automatically* allow GPU sharing.

This paper formulates and solves a scheduling problem that captures the requirements of current clusters and cloud environments, while factoring the features of modern heterogeneous systems. Particularly, we consider both the notion of *urgency* and the concept of *priority*, capturing both through *value functions* associated with different jobs. Specifically, for each job, the value function determines the *"price"* that the user is willing to pay as a function of the execution time. While such differential pricing may not be explicitly supported by the provider, delays captured in our framework can indicate the likelihood of a user dissatisfaction, and thus, their choosing another provider in the future. This, in turn, indicates the likely revenue loss to the provider. In addition, there can be several reasons for service providers to consider priority across jobs and/or users. Our value function captures these priority levels by allowing different maximum values and/or different *decay slopes* to the value functions of different jobs. While our model matches some of the considerations in existing literature [11], [14], [15], [17], [26], [30], [33], [41], [42], the existing work in this area has not considered heterogeneous clusters.

To address this scheduling problem, we propose four novel dynamic mapping and scheduling schemes, broadly categorized as *Uncoordinated* (three schemes) and *Coordinated* (one scheme). These schemes use resource-specific queues and operate by minimizing the imbalance between them. In our coordinated scheme, we allow periodic monitoring of the state of each queue to dynamically readjust the mapping and scheduling at a finer granularity. Uncoordinated schemes lack this periodic monitoring; however, they can dynamically readjust the mapping and scheduling at a coarser-granularity.

Another factor that needs to be considered is related to the nature of these high-end compute resources. Today's processors are massively parallel, i.e. multi-core CPUs typically have up to 16 cores, and GPUs typically have up to 512 cores. It is feasible and desirable to co-schedule multiple jobs on a single resource. Our earlier work [36] has demonstrated effective mechanisms for sharing a GPU in a cluster or cloud environment, using system throughput as the metric of interest. In this paper, we aim at maximizing system profit/user satisfaction, and develop a novel scheduling heuristic to allow jobs to efficiently share multi-core CPUs and GPUs. In particular, we observe that some jobs are less urgent, and/or exhibit performances that do not scale well beyond a certain number of cores, and hence are preferred candidates for sharing without incurring significant penalties.

To demonstrate the effectiveness of our schemes in a realistic setting, we have implemented a prototype scheduler for heterogeneous clusters that include multi-core CPUs and GPUs. Our framework consists of a *Cluster Level Scheduler* and multiple *Node Level Schedulers*. The former runs on the master node, whereas each of the latter runs on compute nodes and is integrated with a runtime consolidation framework that enables GPU sharing. In order to evaluate our scheduling schemes, we have compared them against two baseline schemes, which include the scheduling policy used in TORQUE [10] and another published scheme [34]. Our results show that under varying conditions, we can reduce the *completion time* by up to 22% and improve the *aggregate yield* by a factor of 8.8 times. Furthermore, efficient resource sharing allows an additional 23% gain in the yield.

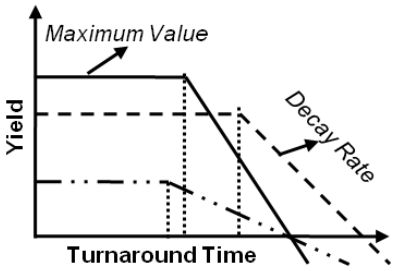


Fig. 1. An Example Value Function (Linear Decay Slope)

II. Problem Definition

We start by explaining the notion of a value function, and relating it to the considerations involved with scheduling on clusters and cloud environments. We then formulate our scheduling problem.

A. Value Function

The *value function* of a job is a user or system defined function that specifies the value, utility, or yield of that job, based upon the *turnaround time*, i.e. the difference between the time of completion of the job, and the time the job is initially submitted. A simple illustration of our value function is shown in Figure 1. In this example, we have *linear decay* in the value function, similar to an earlier formulation [26]. Each job is associated with a *maximum value* which is obtained when the job completes within a specified time limit. If the job is delayed beyond the given time limit, then the value associated with executing the job decays as dictated by the function (decay rate). At each job completion, an *yield* is obtained from the job. The value, utility, or yield could be viewed from the perspective of both the user and the system provider.

The value function combines the notion of *importance* and *urgency* of jobs. It can also represent the Service Level Agreement (SLA) between a user and a service provider. From the users' perspective, the yield indicates the user satisfaction associated with the execution of the job. On the other hand, from the system perspective, the yield indicates the price that the user will pay for such a quality of service. The overall goal of the system is to maximize the aggregate yield across all jobs entering the system. Consequently, the value function can be used by the scheduler to determine the relative priority of the jobs.

We now discuss how the value function we consider captures existing (or desired) scheduling considerations that apply in supercomputing centers and cloud environments. A large number of supercomputing centers within commercial and academic organizations use the Moab scheduler [3]. The commercial enterprise edition of the Moab workload manager allows for configuring different SLAs between the users and the system. We believe that these SLAs can be captured by different value functions. As an example, an SLA might be that X% of the jobs from the user will be finished without a delay higher than Y. Such an SLA can be captured by a value function that steps the value (or yield) down to 0 after a delay of Y. Note that due to the proprietary nature of Moab, the details of underlying scheduling schemes are not available. However, Moab clearly still has the last two of

the three limitations of TORQUE that we pointed out in the previous section.

Cloud environments support a *pay-as-you-go* model. Resource utilization is a key factor for cloud providers, and it is well-known that, within cloud environments, resource *over-subscription* (or under-provisioning) is a commonly used technique to reduce the infrastructural costs[2]. Over-subscription implies that N purchased cores may be physically mapped to M actual cores, where ($N > M$). While scheduling different tasks on potentially over-subscribed resources, cloud providers need to pay attention to the satisfaction of different classes of users. For example, cloud providers often offer *reserved instances*, and owners of these resources will not like to see the performance of their jobs suffer due to over-subscription. At the same time, a cloud environment may have *free* users (e.g. academic users who have been given free credits for courses or research activities) and/or users who have purchased discounted *spot* instances. A cloud provider does not need to be so concerned about the satisfaction of these users. Thus, a value function based formulation that uses different decay functions can capture the service provider's objective in a cloud environment with different classes of users.

B. Problem Formulation

The problem that we address can be stated as follows. *Given a heterogeneous cluster with each node comprising a multi-core CPU and a GPU, we want to schedule each job on either one of the multi-core CPUs or one of the GPUs available in the cluster, so as to maximize the aggregate yield obtained over all jobs entering the system*.

Before showing how we address this problem, we want to clarify two key aspects of our formulation. First, each job will either be *allocated* a multi-core CPU or a GPU, and not both. However, when a job is allocated a GPU for execution, one additional CPU core is also allocated to the job. This is because GPUs are handled as accelerators and its computations are launched from within a CPU thread (also called *host* thread), and the results are transferred back to the CPU upon completion of the GPU task. In section V-D, we evaluate the performance degradation due to sharing one CPU core between a CPU and a GPU job running on the same node. Moreover, there are no established standards for developing applications to use all cores in a multi-core CPU and a GPU at the same time. This is further confirmed through the nature of applications available in popular heterogeneous benchmark suites like Rodinia [16] and Parboil [5]. It has also been observed that most jobs run more effectively either on a GPU or on a multi-core CPU, and only marginally better when trying to use both [13], [18], [32], [35], [38], [45]. Thus, dedicating an entire node (unless otherwise required) to a single application at any given time may lead to a relatively poor resource utilization. Our proposed schemes focus on single-node jobs, but can be extended to other configurations (multi-node, multi-GPU jobs etc) in the future. However, it should be noted that recent logs from a supercomputer center [7] (see chart in 2nd row and 1st column), show that jobs with moderate parallelism form a substantial fraction (almost 80%) of the entire workload.

The second key aspect of our problem formulation is that each submitted job can be *flexibly scheduled* either on the multi-core CPU or the GPU. This assumption is justified by the following considerations. First, OpenCL [4] is becoming a preferred choice of programming model for heterogeneous architectures, and is device agnostic. Using OpenCL, an application can be written once and compiled for many devices to produce different binaries. Second, other commercial compilation products like PGI CUDA-x86 [6] transform CUDA into optimized x86 compatible code. Furthermore, MCUDA [43] performs transformations of CUDA code into C, Ocelot [19] and SWAN [25] performs a CUDA-to-OpenCL transformation, whereas OpenMPC [31] allows OpenMP-to-CUDA translation. Clearly, the community is moving towards a direction where the application developer can write a program once and flexibly generate code for different target platforms. Thus, our scheduling formulation actively exploits such portability to improve efficiency.

[2]For example, see http://cloudscaling.com/blog/cloud-computing/subscription-modeling-cloud-performance

III. Scheduling Heuristics

In this section, we give a detailed description of various scheduling heuristics we have developed for the problem described in the previous section.

A. Overall Approach

The overall approach is as follows. The jobs are submitted in batches and the arrival rate of the jobs are not known a priori. Each submitted job is added into either the CPU queue or the GPU queue. The jobs within each queue are sorted for scheduling based on a criteria we will describe in section III-B. Under normal conditions, jobs from a particular type of queue (CPU queue) will be executed on the corresponding resource (multi-core CPU). But, because imbalance between the queues is likely, a resource type (for instance CPU) may have an empty queue, while it is free. To address such imbalance across queues, we use a number of schemes to dynamically move a job from one queue to another (as described in section III-C). Note that we do not allow *pre-emption* of an already executing job. This assumption is justified by the overheads and lack of software technology to pre-empt a GPU job. Practical batch scheduler like TORQUE [10] use the *walltime* (parameter of the *qsub* command) from the user to estimate how long the job will occupy the resource and, consequently, to guide scheduling decisions. Techniques have been proposed for system-generated prediction [46], however, still relies on user-estimates for scheduling. Similarly, our scheduler expects a *walltime* for each type of resource (i.e., CPU and GPU). We refer to them as optimal walltime (*OptimalWT*) and non-optimal walltime (*Non-OptimalWT*) corresponding to the optimal and non-optimal resource respectively.

B. Initial Placement and Ordering

When a batch of jobs enters the system, we first determine the resource type (CPU or GPU) where each job runs with the optimal walltime, i.e. the optimal resource (type) for the job, and enqueue it into the corresponding resource queue. The queue in which the job gets initially enqueued into is referred to as *Optimal Resource Queue* for that job, whereas the other queue is referred to as *Non-Optimal Resource Queue*. Next, the jobs in each queue have to be ordered to maximize the aggregate yield. For this purpose, we use a *market-based* scheme (referred to as *Reward*), which can be viewed as an adaptation of earlier work from Irwin *et al* [26] to our problem formulation (particularly the fact that we are targeting heterogeneous clusters).

Recall from our value function (as shown in section II-A) that the *yield* of the job i is the value obtained from the job when its execution is finished, and is given by:

$$yield_i = max_value_i - (delay_i \times decay_i) \quad (1)$$

Here, max_value_i, $delay_i$, and $decay_i$ are the maximum possible value (or yield) achievable from the job, the delay in finishing its execution, and the decay rate of the value for this job, respectively. Note that, the *delay* factor, in turn, could be a sum of four penalty components: 1) Queuing delay, 2) Slowdown due to sharing a CPU core between a CPU job and GPU job running simultaneously within the same node, 3) Slowdown in execution if the job is scheduled on a non-optimal resource, and 4) Slowdown due to sharing a resource, if its execution is shared with another job.

The Reward scheme for ordering jobs is based on two terms: *present value* and the *opportunity cost*. Present Value (PV) from running a job can simply be defined as the value (or yield) that could be obtained after time t, which is the optimal walltime of the job, but after discounting the risk involved in running this job. For instance, if two given jobs have same gain, then it might be preferable to run the job with the shorter execution time first. This is because, the shorter job will have a lower risk of deferring an urgent job that may arrive later. Formally, the present value (PV) of a job i per unit time is defined as follows.

$$\frac{PV_i}{OptimalWT_i} = \frac{yield_i}{(1 + discount_rate \times OptimalWT_i)} \tag{2}$$

Note that $discount_rate$ is a tunable factor, which can be chosen based on how significant this term should be.

Next, the second component of the Reward heuristic considers the losses from the opportunity cost of selecting a job i over another job j in the queue. The idea is that if the job j has a higher decay rate than the job i, then it may be preferable to run the job j ahead of the job i, even if it has a lower value. Thus, the opportunity cost of choosing to run the job i is defined as the aggregate degradation caused in the yield of other jobs in contention at the same time. Formally, the opportunity cost, $cost_i$ of the job i is defined as:

$$\frac{cost_i}{OptimalWT_i} = \sum_{j=0}^{n} decay_j - decay_i \tag{3}$$

This summation is performed over all other jobs that are currently in the queue. Overall, giving equal weight to the two factors shown in Equations 2 and 3, the *Reward* is computed for each job in each queue as follows:

$$Reward_i = \frac{(PV_i - cost_i)}{OptimalWT_i} \tag{4}$$

After this, the jobs within each pending queue are ordered on to the corresponding resources in the order of the *Reward* score.

C. Scheduling Schemes

We have so far shown how we separate the jobs into two different pending queues (CPU and GPU), and how to order each queue. The remaining questions are, *when* and *which* jobs could be moved from one queue to another. We have developed a number of schemes for dynamically moving a job from one queue to another.

The motivation for moving jobs from one queue to another is as follows. Whenever a resource becomes free, and if the corresponding queue is empty, the resource might stay idle, unless a job from the other queue is selected for execution. Allowing a resource to stay idle reduces utilization, and is not desirable. Moreover, this means that jobs in the other queue are being delayed unnecessarily, and thus their yield is being lowered. However, now the challenge is in deciding which job should be moved from their optimal resource queue to execution on the non-optimal resource. Note that different jobs may have different amounts of slowdown when they are moved for execution on the non-optimal resource. We have developed three different *uncoordinated schemes* to select jobs for execution on the non-optimal resource.

Furthermore, it is interesting to note that even if a resource queue is not empty, there could be a large difference in the amount of delays being incurred by jobs in the two queues. Such an imbalance will degrade the yield of one queue relative to the other, unless there is a coordination between the two queues. To address this problem, we have also developed a *coordinated scheme*, which can execute job(s) on non-optimal resources even if they are not idle.

1) Uncoordinated Schemes: In the case of uncoordinated schemes, a job is moved from the optimal resource queue to the non-optimal resource queue *only when the non-optimal resource is free and its job queue is empty*. Otherwise, there is no coordination between the queues. For uncoordinated schemes, there could be multiple factors based on which a job can be selected for execution on the non-optimal resource queue. The schemes we have developed are based on two key questions that can help maximize aggregate yield: 1) Which of the jobs in the optimal resource queue, when moved, will have the *best reward* on non-optimal resource?, 2) Which of the jobs in the optimal resource queue, when moved, will *suffer* the *least reward penalty* on non-optimal resource? Based on these two factors, we have developed three new heuristics.

Last Optimal Reward (LOR): Recall that after initial placement of jobs (discussed in Section III-B) in each queue, we computed Reward (Equation 4) for each job in each queue specific to its resource. These queues are sorted based on the Reward factor. Our first scheme, Last Optimal Reward, exploits this reward information. Suppose that a CPU becomes free, but the CPU queue is empty. So, we want to steal a job from the GPU queue to execute on the available (idle) CPU. Last Optimal Reward scheme chooses the last job from the GPU queue, to be executed on the CPU, i.e. the job that will have the *least reward score* on the optimal resource queue.

The reasoning behind this scheme is as follows. The last job in the optimal queue will anyway produce the least reward (or may be less urgent), among all current jobs, in its optimal resource. Thus, one could argue that there is likely the lowest amount of risk involved in moving this job into the non-optimal resource. Moreover, this job is likely to be executed much sooner in this case. What this simple scheme does not consider, however, is the non-optimal penalty of the job to be moved with that of the other jobs in the queue. Therefore, if the last job happens to have a high non-optimal penalty, then this scheme can possibly suffer more loss than the yield.

First Non-Optimal Reward (FNOR): In contrast to the previous scheme (*LOR*), *FNOR* exploits the reward the job could produce when executed on the non-optimal resource. Note that this scheme is very different from *LOR* as it explicitly considers the non-optimal penalty of the job. Thus, *FNOR* is based on the rationale that it is beneficial to move a job that will have the highest reward score on the non-optimal resource. In order to compute the reward score on the non-optimal resource, we combine the existing reward score of the job for optimal resource, with the penalty the job will suffer, if executed on the non-optimal resource . For each job i, we

first determine the *Sufferage Factor*, defined as,

$$Sufferage_Factor_i = \frac{Non - OptimalWT_i}{OptimalWT_i} \quad (5)$$

Next, we compute the *Non-Optimal Reward* score, by combining the optimal *Reward* shown in Equation 4 and the sufferage factor, as follows:

$$Non - OptimalReward_i = \frac{OptimalReward_i}{Sufferage_Factor_i} \quad (6)$$

Note that sufferage factor is always greater than 1, hence, the *Non-Optimal Reward* score is always less than the *Optimal-Reward* score.

Finally, the job that gives the highest Non-Optimal Reward is chosen to be scheduled on the non-optimal resource. Consequently, if there are two jobs that suffer similar penalties when moved to the non-optimal resource, then this heuristic prefers the job that will result in the better reward, which corresponds to the more valuable and/or the more urgent job.

Unlike the previous scheme, there is more computational cost associated with this scheme. The complexity for computing the *Non-Optimal Reward* for each job in a queue of length N is $O(N)$, and moreover, the complexity of choosing the job with the highest Non-Optimal Reward is $O(Nlog(N))$.

Last Non-optimal Reward Penalty (LNORP) In this scheme, we take the scheduling considerations one step further when compared to our previous scheme, *FNOR*. Specifically, while *FNOR* gives preference to the job that has the highest reward score on the non-optimal resource, it fails to consider the reward score degradation experienced due to non-optimal penalty. Thus, if there are two jobs that have similar *Non-Optimal Reward* score, unlike *FNOR*, *LNORP* will prefer to move the job that will have the lesser reward penalty. The *Non-Optimal Reward Penalty (NORP)* for the job i, is computed from both *Non-Optimal Reward* and the *Optimal Reward*, which were shown earlier in Equations 6 and 4, respectively. NORP is calculated as follows:

$$NORP_i = OptimalReward_i - Non - OptimalReward_i \quad (7)$$

This heuristic can be better than *FNOR* when there are jobs with similar rewards, but vary in their degree of penalty on the non-optimal resource. Similar to *FNOR*, this scheme also has a complexity of $O(Nlog(N))$ for a queue size N.

2) Coordinated Scheme: The three uncoordinated schemes described above consider moving a job to a different queue *only* when a resource is idle. These schemes can be effective in improving the aggregate yield by handling the imbalance and reducing the idle time of the resources. However, interestingly, in some scenarios, it may be *necessary to move a job from one queue to another, even though neither of the resources is idle*. Consider a case when there is an imbalance in the urgency of the jobs between the two queues, i.e, one queue may be skewed towards less urgent jobs, while the other queue is skewed towards more urgent jobs. Then, the jobs in one queue will decay much faster than the other, rendering uncoordinated schemes ineffective due to loss of the global view of both the queues. This consideration leads to a new *coordinated* scheme.

We have developed a coordinated scheme that decides when to move a job from one queue to another, even when no resource is idle and when neither queue is empty. This scheme has two components: (i) A logic for detecting when coordination is required between the two queues, and (ii) As a result of coordination, a heuristic for which job to move into the non-optimal queue.

First, we describe the factor based on which coordination is invoked. Consider the causes of imbalance in the queue that makes one resource idle while the other is busy. Primarily, it is either because the number of jobs in one queue is more than the other, or that the length of execution of the jobs is much longer than the other. This can be captured by calculating the sum of the queuing delay for each job in the entire queue. Another potential factor causing imbalance is the decay rates across all jobs in each queue. Thus, in a case when imbalance is due to the queuing delay and/or the decay rate, then the yield of jobs in the heavy queue will be affected significantly. Hence, we introduce **Total Queuing-Delay Decay-Rate Product (*TQDP*)** as a metric for detecting when such coordination between queues is desirable. Formally, for each job j and for each queue i, we calculate *TQDP* as follows.

$$TQDP_i = \sum_{j=0}^{n} Queuing_delay_j \times decay_j \quad (8)$$

Note that the *queuing_delay* for each job is propagated based on its position in the queue. If the computed *TQDP* varies significantly between the two queues, then, even if both the job queues are non-empty, a job from the heavy queue is executed on the non-optimal resource, until it reaches an acceptable imbalance *threshold*. The threshold for such imbalance can be selected by the system. After this, the scheduler again resumes normal operation, where each resource is assigned a job only from the queue corresponding to its resource type. The critical advantage over the uncoordinated schemes is that such a fine-grained mechanism provides quicker attention to the rapidly decaying jobs, and consequently better user satisfaction.

TQDP can only indicate if there is an imbalance between the two queues. However, we still need to decide which job should be moved from the optimal queue to the non-optimal queue. For this purpose, we use the *Least Penalty* heuristic, i.e., we select the job that will have the least *Sufferage_Factor* on the non-optimal resource, as calculated through Equation 5. Overall, we refer to this coordinated scheme as the **Coordinated Least Penalty *(CORLP)*** scheme.

D. Heuristics for Sharing

In this subsection, we describe our heuristic for sharing and the logic for determining when sharing will be valuable.

Our scheduler considers the possibility of having up to two jobs share a single resource (multi-core CPU or GPU) at any given time. In particular, we only allow *space-sharing* and do not let the jobs time-share the processor cycles. For instance, on a CPU with 8 cores, two jobs may each use 4 cores from the same CPU. Similarly, two jobs may each use 7 of the 14 SMs in a Tesla C2050/C2070 GPU. Our decision is motivated by the observation that space-sharing does not create a significant resource contention, whereas time-sharing may in some situations adversely impact the performances due to increased resource contentions [36].

The tradeoffs associated with sharing a resource is as follows. First, from the view-point of a single application, as it gets half of the resources, it is likely to incur a slowdown, which will impact the yield it obtains. Applications, as we can expect, have varying degree of scalability. We assume that through analytical models [27], [28] and/or profiling tools [23], it is possible to have a reasonable prediction of the scalability of a given application on a multi-core or many-core architecture. Thus, we assume that applications are

characterized into three groups, i.e., those with low, medium, and high scalability.

Now, a job with a very low scalability beyond half the number of cores will have a low penalty due to sharing, whereas, a job with a good scalability beyond half the number of cores will have a high penalty. Thus, from the view-point of the aggregate yield, it is not preferable to share jobs for which yield decays rapidly, since yield gain for the other job sharing the resource will at most result in the same aggregate yield. In addition, it is also necessary to consider the overall load in the system. If the load is high, it will be desirable to have two low decay and/or low scalability jobs share one resource, so that another resource is available for a job that is highly scalable and/or has high priority.

Based on the above observations, our scheme for deciding *which* jobs should share a resource is as follows. Whenever a new batch of jobs arrives, after its initial placement (as described in section III-B) into the queue, we calculate the predicted yield of the jobs based on its current position in the pending queue. If there are jobs with a very low or even negative yield, we take this as an indication of heavy load and enable sharing. Next, in each queue containing jobs with such low yield, we find the candidate jobs and mark them for sharing. However, in order to find the right candidates for sharing we need to consider both scalability factor and the decay rate of the job. Thus, we propose the concept of **Scalability-Decay Factor**. We first group the jobs under consideration in the order of low, medium and highly scalable jobs. Next, within each group, the jobs are sorted in the ascending order of their decay rates. Such an ordering of jobs ensures that sharing is preferred for low scalable jobs and low decay jobs compared to highly scalable and high decay jobs respectively. However, in order to decide *how many* jobs to share, we parametrize the heuristic with a *Sharing Factor*. This factor dictates the fraction of jobs from the top of the pending queue that will be marked for sharing.

IV. Software Framework Design

In this section, we provide details on the design and implementation of our scheduler.

A. High-Level Scheduler Framework Design

The high-level framework design of our scheduling system is shown in Figure 2(a). Our scheduler follows the classic master-slave architecture, and consists of two major components: Cluster Level Scheduler (CLS) and Node Level Schedulers (NLS). The functionalities of these components are detailed below. We assume a shared file system across all nodes in the cluster. This is consistent with the configuration of most clusters and supercomputers in use today.

Cluster Level Scheduler (CLS): The CLS is installed on the master node, and has three responsibilities: (i) management and monitoring of the job queues, (ii) generation of scheduling decisions according to the heuristics described in Section III, and (iii) communication of the scheduling decisions to the compute nodes.

The CLS maintains four types of job queues: Submission, Pending, Execution and Finished queues. All the jobs that are submitted by the users are placed into the submission queue. Since each job can be executed on a GPU or on a multi-core CPU, we maintain separate queues for these two types of resources. The order of scheduling of the jobs will depend on the pending queue. The execution and the finished queues are used for maintaining the status of the job, and for later analysis of the jobs' execution. While maintaining two separate queues, we have two design choices. One possible design is to enqueue all the jobs in both the CPU and the GPU queues. This allows the flexibility of choosing a job from either queue. However, this implementation leads to the challenge of maintaining consistency among the queues, which can be expensive, and will not scale with the size of the cluster, the number of resource types, and the number of submitted jobs. Alternate design is to place a job into a single queue (either the CPU or the GPU queue), thereby eliminating the need for maintaining consistency. This implementation requires dynamic schemes to move jobs between queues, which can be expensive in case of uneven load distribution (as described in sections III-C1 and III-C2). However, we chose this design as the frequency of such dynamic reassignments will be small.

The core functionality of the CLS is to perform scheduling decisions through the heuristics described in Section III. Once the scheduling decisions are made, they will be communicated to the appropriate compute node by sending job-specific metadata over a TCP communication channel. To allow the CLS to maintain an up-to-date view of the resource utilization within the cluster, each compute node will notify the master node upon job completion.

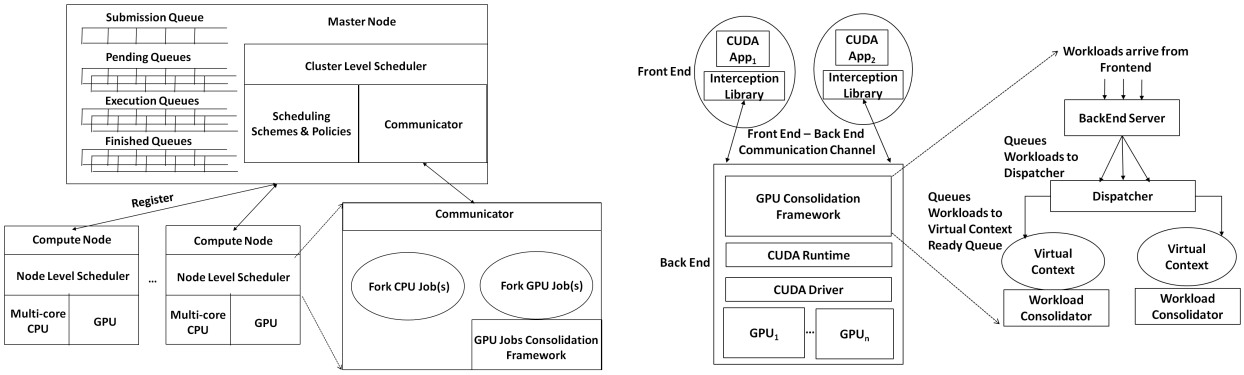
Node Level Scheduler (NLS): The NLS is installed on all compute nodes in the cluster. Each compute node, whenever added to the cluster, registers itself with the master node. The NLS will fork a new process for each job assignment received by the CLS. In case of sharing, multiple CPU jobs will concurrently run on the compute node using the scheduling mechanisms provided by the underlying OS, and will thereby share the multi-core CPU. On the other hand, GPU sharing requires an additional software mechanism. To this end, we use the GPU runtime consolidation framework described below.

B. GPU Sharing Framework

We now briefly describe the GPU runtime consolidation framework used to enable GPU sharing within the NLS. The interested reader may refer to our recent work [36] for more details on GPU sharing mechanisms.

Until recently, task parallelism was not supported on NVIDIA GPUs (unless hard-coded within a single GPU kernel). It is only starting from GPUs with compute capability 2.0 (Fermi cards and later) that NVIDIA introduced a *concurrent kernel execution* feature, thus enabling task parallelism across GPU kernels. This feature, offered by CUDA, is limited to kernels launched within a single process context, and therefore does not allow GPU sharing across distinct applications. To overcome this limitation, our software consolidation framework introduces the concept of *virtual context*, that represents an interface between the applications and the CUDA runtime. By mapping multiple applications onto the same virtual context, it is possible to enable the concurrent kernel execution feature offered by CUDA across application boundaries.

Virtual contexts provide a *mechanism* to enable GPU sharing across applications. However, GPU sharing is not always beneficial, and policies to dynamically determine which applications are suitable for sharing are required. To this end, we note that, similarly to CPUs, GPUs allow two kinds of sharing, namely *space*- and *time-sharing*. In the former case, co-scheduled kernels will be allocated a subset of the available streaming multiprocessors (SMs); in the latter, co-scheduled kernels will use the same computational resources during distinct compute cycles. Intuitively, space-sharing leads to less interference between applications (conflicts are mostly limited



(a) High-level Scheduler Framework Design

(b) Runtime Consolidation Framework For GPU Sharing

Fig. 2. Scheduler Framework Design and its Components

to the use of the available memory bandwidth), and is therefore preferable compared to time-sharing. Therefore, in this paper we restrict ourselves to space-sharing.

Our proposed GPU consolidation framework is represented in Figure 2(b). The framework relies on two software mechanisms: *CUDA calls' interception and redirection* and *virtual context management*. CUDA calls' interception allows handling applications at runtime, without any source code modifications. In particular, our framework leverages the interception mechanism provided by the open source project gVirtuS [24]. Following the split-driver model, CUDA calls are intercepted by a frontend library, and redirected to a backend demon (or server) running in the same OS space as the CUDA runtime. The core functionalities of the GPU consolidation framework are implemented within two backend components: the Dispatcher and the Workload Consolidator. The Dispatcher inspects CUDA calls originated within the applications and redirected by the frontend library to the backend server. In the original design [36], the Dispatcher makes consolidation decisions based on the analysis of the execution configuration (number of thread blocks, number of threads, and shared memory usage) of the kernels. However, *in our current proposal all scheduling (and consolidation) decisions are made by the CLS*. Therefore, *the Dispatcher will simply modify the execution configuration of the kernels to be consolidated in order to force space-sharing*. In particular, it will set the number of thread blocks to half the number of available SMs. In addition, if the per-thread-block register and shared memory usage allows it, it will increase the number of threads per block so to have a total number of threads as close as possible to that in the original execution configuration. Next, the Workload Consolidator uses virtual contexts to allow concurrent kernel execution among different applications. In particular, the Workload Consolidator runs each kernel within a different CUDA stream, and defers synchronization calls until all consolidated kernels have completed execution. The virtual contexts created by the Workload Consolidator issue CUDA calls associated to them to the underlying CUDA Runtime.

V. Experimental Results

We first describe our evaluation methodology, and then present the results.

A. Evaluation Methodology

In this subsection, we provide details on our cluster hardware, benchmark applications, experimental setup, and the baselines we have considered in our evaluation.

Cluster Hardware: Our experiments are conducted on a cluster consisting of 16 heterogeneous nodes. Specifically, each machine is equipped with two Intel Quad-core Xeon E5520 CPUs (total of 8 cores, each running at 2.27 GHz), 48 GB main memory and an Nvidia Tesla C2050 (Fermi) GPU card. Each Tesla C2050 card has 14 streaming multiprocessors (SMs), each containing 32 cores (448 cores overall) running at 1.15 GHz, and 3 GB device memory. The machines are connected through an Infiniband network.

Benchmarks: In our evaluation, we use ten popular benchmark applications taken from different domains like scientific computations, financial analysis, data mining, and machine learning. These applications are listed in Figure 3. For each of them, we consider three different execution configurations: small dataset, large dataset, and large number of iterations (on large dataset). In Figure 3, we show the results obtained by running each benchmark program with these three configurations. Specifically, we report the sequential execution time (on CPU), as well as the GPU and multi-core CPU speedups relative to sequential execution. We now analyze the performance trends in these benchmarks. When small datasets are used, the majority of the benchmarks (7 out of 10) perform better on the multi-core CPU as compared to GPU. The trend changes when large datasets are used; in this case, the majority of the benchmarks (6 out of 10) run faster on the GPU. Two observations can be made from these results: first, the use of small datasets leads to a low GPU utilization; second, the initial GPU overheads are not well amortized over short execution times. In addition, when the number of iterations increases (third configuration), the ratio of data transfer overhead to the computation time decreases, leading to increased performance improvement on GPU (e.g. BlackScholes and MonteCarlo) as compared to CPU.

Experimental Setup and Baselines: Unless otherwise specified, our experiments have been conducted using a total of 256 jobs scheduled on our 16 node cluster. To the best of our knowledge, job traces submitted on heterogeneous (CPU-GPU) clusters are not publicly available. Thus, for our experiments, we generated a number of different workloads, by randomly selecting each job from the pool of 30 jobs described earlier. For the purpose of our experiments, we assume that the maximum value of each job is equivalent to its optimal walltime. In addition, the value of the job starts decaying beyond the optimal walltime. We have used a bimodal distribution of decay rates [17], [26]. Note that each of the data point has been

	Small Dataset				Large Dataset				Large No. of Iterations			
Benchmarks	Seq. CPU Exec. (sec)	GPU Speedup (GP)	Multicore Speedup (MP)	Data set Characteristics	Seq. CPU Exec. (sec)	GPU Speedup (GP)	Multicore Speedup (MP)	Data set Characteristics	Seq. CPU Exec. (sec)	GPU Speedup (GP)	Multicore Speedup (MP)	Data set Characteristics
PDE Solver	1.8	3.8	7.1	7168*7168	7.3	4.7	6.8	14336*14336	722.1	4.3	8.1	14336*14336
Image Processing	8.4	5.6	7.5	7168*7168	33.8	5.1	7.8	14336*14336	3385.5	4.8	8.0	14336*14336
FDTD	2.1	1.3	7.7	7168*7168	8.4	2.2	7.6	14336*14336	423.3	1.8	7.9	14336*14336
BlackScholes	0.7	0.6	6.8	2.5 mil options	2.6	2.1	7.2	10 mil options	269.1	92.8	7.8	10 mil options
Binomial Options	3.0	2.3	4.2	128 options	11.8	5.6	4.2	1024 options	1213.6	12.2	4.3	1024 options
MonteCarlo	11.0	9.4	7.9	256 options	45.4	38.4	7.9	1024 options	453.3	368.5	7.8	1024 options
Kmeans	74.2	6.3	7.7	0.4*10 ^ 9 points	330.0	12.1	7.8	1.6 * 10 ^ 9 points	1593.8	12.6	7.9	1.6 * 10 ^ 9 points
KNN	16.8	2.9	6.2	16777216 points	67.3	7.8	6.2	67108864 points	1691.1	58.4	6.9	67108864 points
PCA	33.8	9.1	5.6	65536*80	142.0	9.7	5.6	262144*80	2835.7	11.8	6.2	262144*80
Molecular Dynamics	6.7	12.8	7.3	32000 nodes, 3968000 edges	46.6	12.9	7.9	256000 nodes, 31744000 edges	593.8	20.8	7.8	256000 nodes, 31744000 edges

Fig. 3. Benchmarks Characteristics

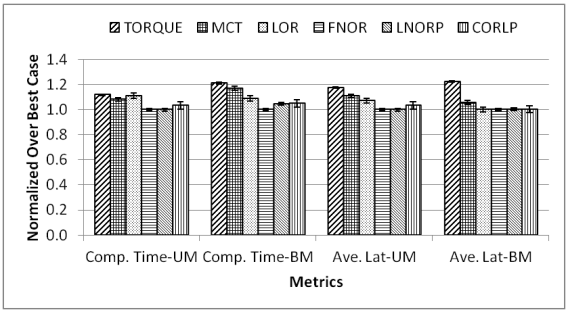


Fig. 4. Comparison of Scheduling Schemes Based on Completion Time and Latency (Lower is Better)

collected after running the corresponding experiment for three times. In all cases the observed variation is less than 2.5% We compare our proposed scheduling schemes with two existing policies: a practical scheme used by the current distribution of TORQUE [10], a popular scheduler for today's heterogeneous clusters, and a published scheme, called Minimum Completion Time [34]. We refer to these two schemes as *TORQUE* and *MCT*. *TORQUE* schedules by always assigning the job to the optimal resource. If the optimal resource is not available, then it waits until the requested resource becomes available. *MCT* assigns a job to the resource where it will complete the earliest depending on the waiting times on each resource.

B. Comparison with TORQUE and MCT

Comparing our schemes with traditional scheduling schemes, like TORQUE and MCT, is challenging, because the two sets of approaches focus on different metrics. TORQUE and MCT are designed to reduce the completion time and latency, whereas our schemes try to maximize the average yield. Thus, we have compared the two sets of schemes using completion time and latency, as well as average yield.

We first compare all the scheduling schemes based on completion time (referred as *Comp. Time*) and average latency (referred as *Ave. Lat*), which are the TORQUE-based metrics. The basic idea is to see how the value-based scheduling schemes perform with respect to the metrics that TORQUE is designed for. The results are shown in the Figure 4. In this experiment, we use two different job mixes: Uniform Mix

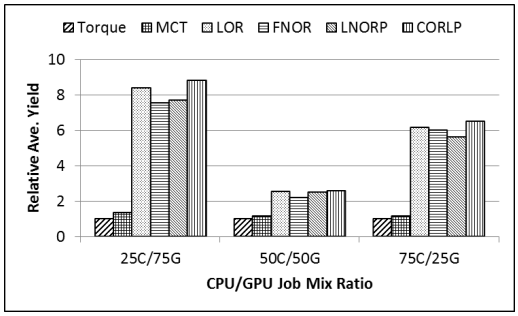


Fig. 5. Impact of Varying CPU-GPU Job Mix on Yield (Higher is Better)

(*UM*) and Biased Mix (*BM*). In *UM*, the jobs are uniformly distributed across CPU and GPU queues, whereas in *BM* the jobs are distributed with a bias (25%-75%) to one particular resource queue.

First, consider the *Comp. Time* metric. For the *UM* case, we can see that *LOR* performs only as good as *TORQUE* as it is a very simple scheme and does not consider the penalty due to non-optimal assignment. However, more intelligent schemes like *FNOR* and *LNORP* perform about 10% better than *TORQUE*. This is because the benefit from efficiently using the resources (zero idle time) outweighs the penalty due to the non-optimal assignment. In fact, the inefficiency due to resources being idle is further aggravated in the *BM* case for *TORQUE*. We can see that *FNOR* and *LNORP* can outperform *TORQUE* by about 22%. For both *UM* and *BM*, on the average, *TORQUE* leaves each of the multi-core CPU and the GPU idle for 5% and 14%, respectively. Next, consider the *Ave. Lat* metric. For both *UM* and *BM*, our schemes outperform the baselines by about 20%. There are three reasons for this. First, our schemes may order the job queue with a preference for short jobs (see Section III-B). Next, we do not leave any resource idle. Finally, we can minimize the non-optimal penalty using schemes like *LNORP*.

We now focus on the value-based metric, the *Average Yield*, for the rest of our evaluation. In our first experiment focusing on this metric (Figure 5), we create three job mixes: GPU skewed (*25C/75G*), Uniform (*50C/50G*), and CPU skewed (*75C/25G*). Note that GPU skewed and CPU skewed job mixes will create imbalance in the resource queues. The results are shown in the Figure 5. The Y-axis shows the *Average Yield*, which is our metric. From the results, we can see that our schemes outperform (up to a factor of 8.8) both *TORQUE* and *MCT* for all job mixes. It is interesting to note that even with uniform mix our schemes outperform the baselines by a factor of 2.3. There are multiple reasons as to why our schemes give significant yield improvement over the baselines. First, the baseline schemes do not have the notion of value. Next, *TORQUE* performs a static mapping of jobs to resources and, as a consequence, leaves some resources idle even when they are available. On the other hand, our proposed schemes use the notion of value to prioritize the jobs, flexibly schedule them, and eliminate the idle times, thus improving the overall yield. Note that the improvement in yield is higher for skewed job mixes compared to the uniform job mix as a result of larger resource idleness with *TORQUE*.

The previous experiment used a linear step function. To show the generality of our schemes, we now study the performance under two different value decay functions, i.e. besides the linear decay function, we also use a *step decay* function. In the step decay function, a job will achieve maximum value

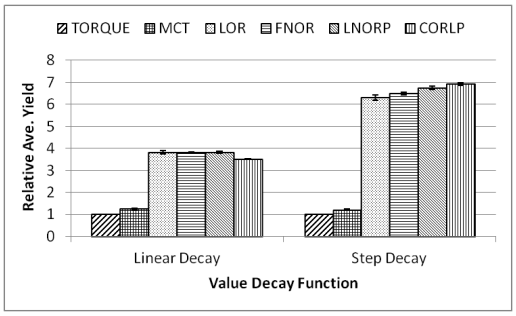


Fig. 6. Impact of Different Value Decay Functions on Yield (Higher is Better)

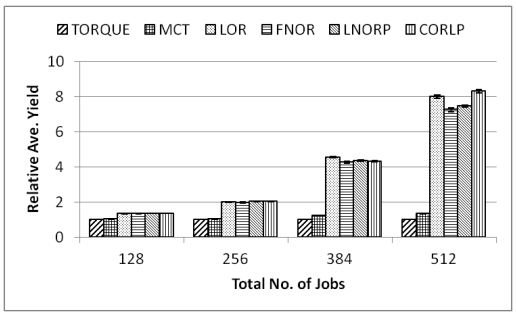


Fig. 7. Impact of Increasing Load on Yield (Higher is better)

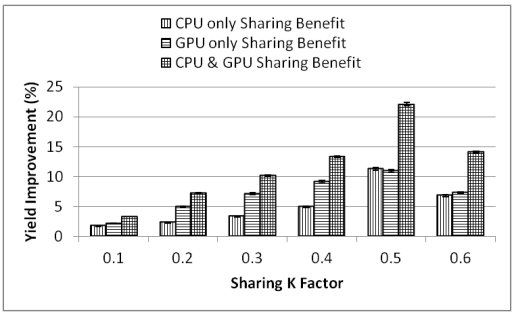


Fig. 8. Impact of Varying K-Factor on Sharing (Higher is Better)

No. of Jobs	Yield Improvement (%)
128	18.2
256	20.1
384	22.3
512	22.9

Fig. 9. Benefit from Sharing with Increasing Load (Higher is Better)

if the job is completed within a specific time limit, but beyond that, for every step (which is a delay by a certain fraction of the optimal time), the value decreases by a certain fraction. The results from the experiments are shown in Figure 6. We can see that our proposed schemes perform significantly better than the baseline schemes, irrespective of the decay function. Yield improvement is even better with step decay, as compared to the improvements when linear decay is being used (factor of 7 as compared to 3.7). This is because the penalty due to delay in step decay function is more coarse-grained. Overall, results from this experiment show that our scheduling schemes are adaptive to different value functions.

We now study the effect of increasing load (and the corresponding scalability in yield) on these scheduling schemes. The results are shown in Figure 7. In this experiment, we vary the number of jobs from 128 to 512. As the load increases, the gap in the relative yield achieved by the baseline schemes and our proposed schemes increases linearly (a factor of 8.2 at 512 jobs). In particular, beyond 128 jobs, the average yield of *TORQUE* and *MCT* starts to decrease linearly, while, on the other hand, it is observed that our schemes achieve a sustained average yield, despite heavy load. This can be explained by observing that the uncoordinated and coordinated schemes take into account decay rate and value while minimizing the imbalance among resources, and hence, can sustain heavy loads. In most cases, our proposed schemes are comparable, and at 512 jobs, *CORLP* lead to the best yield.

C. Benefit from Enabling Sharing

In this section, we study the effect of resource sharing on the yield. From the benchmarks analysis reported in Figure 3, we found that approximately 30% of the jobs have sub-linear scalability (low and medium) on both CPU and GPU. Thus, in the following experiments, we fixed the fraction of jobs that exhibit low and medium scalability to 10% and 20% (a total of 30%), respectively. As explained in section III-D, the sharing heuristic is guided by two parameters: (i) the tunable *Sharing K Factor*, and the (ii) Scalability-Decay Factor. The former determines how many jobs will execute on a shared mode, and the latter decides the candidates for sharing.

We first study the impact of sharing due to various K factors. The results are shown in Figure 8, where we report the improvement obtained from sharing compared to the best of the four uncoordinated and coordinated schemes. For this experiment, we show the impact of sharing on CPU only, GPU only, and both CPU and GPU. At lower values of K (0.1 and 0.2), the gains from sharing appear marginal. However, as we increase the K factor, the yield improves until K=0.5. At K=0.6, more sharing leads to increasing penalty and hence, benefit starts decreasing. This shows that in both cases, when the amount of sharing is too less or too much, the penalty due to sharing is significant. This experiment emphasizes a careful selection of K value, and shows that at K=0.5, we can achieve an yield improvement of about 22.1% due to sharing.

Another potential advantage of sharing is an added tolerance towards heavy loads. We verify this benefit with increasing loads, and the results are shown in Figure 9. Thus, as we increase the load from 128 to 512 jobs, more jobs are subject to high queuing delays, which impacts the yield of the jobs. Allowing certain jobs to share a resource ensures that more resources will be available for high priority and urgent jobs. Moreover, using our heuristic, only the jobs with low or medium scalability and/or the jobs with low decay rate are preferred for sharing. Overall, we can see that sharing allows a sustained improvement in the yield (up to 23%), as compared to the best of our uncoordinated and coordinated schemes.

D. Overhead of Sharing a CPU Core

When we schedule a job on the multi-core CPU and a job on the GPU simultaneously within the same node, one CPU core will be shared by these two jobs. In this subsection, we evaluate the overhead of sharing a CPU core. The results are presented in the Figure 10. For various job mixes, where each mix containing 256 jobs is scheduled on 16 nodes, we measure the slowdown caused by sharing one CPU core. Note that this does not include other slowdowns due to queuing delay or scheduling on a non-optimal resource. The overheads are

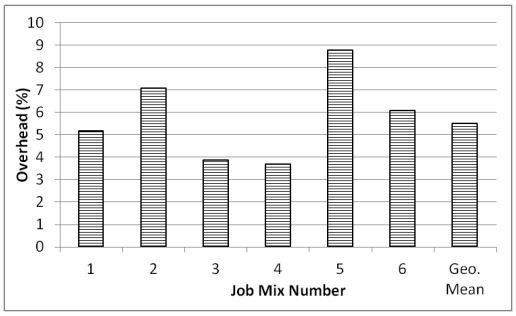


Fig. 10. Overhead of Sharing One CPU Core Between a CPU and a GPU Job

averaged over all the jobs. We observed that for 6 different job mixes, the overhead varies between 3.7% to 8.8% with a geometric mean of 5.5%. The variation in overhead is attributed to the nature of the GPU jobs and its data transfer pattern with the CPU (or host). The overhead increases with the frequency of data transfers between GPU and CPU. Overall, the low overheads observed in our experiment indicate that it is indeed desirable to schedule the CPU and GPU jobs simultaneously on the same node. Moreover, sharing increases the utilization without significantly affecting the overall aggregate yield.

E. Summary of Results

To summarize, using TORQUE-based metrics, we show that our proposed value-based schemes can still outperform TORQUE by about 22%. Using the *Average Yield* (value-based) as the metric, we demonstrate significant gain in the yield (up to a factor of 8.8 times) using the four proposed schemes when compared to the baseline schemes, *TORQUE* and *MCT*, depending on various factors like workload mix, value decay functions, and system load. Finally, we also show that by judiciously enabling sharing of resources we achieve a sustained improvement in the yield of up to 23% even at high system load.

VI. Related Work

In this section, we compare our work with related research efforts. Scheduling for parallel systems has been extensively studied over the past two decades [20], [29], [34], [39], [40], [44], [46], [47]. Here, we restrict our discussion to the work in parallel job scheduling for heterogeneous resources, and to the work along the lines of value-based scheduling.

We first compare our work with the existing schedulers for heterogeneous resources. Torque [10], an open-source resource manager, is being widely used in hundreds of supercomputer centers to manage heterogeneous clusters comprising multi-core CPUs and GPUs. However, the mapping and scheduling strategy used by Torque has been fairly simple. Specifically, while waiting for a resource, it does not consider the possibility of assigning the job to another resource type. Thus, Torque cannot exploit the flexibility that popular framework like OpenCL provides. Other related efforts [21], [34] do consider the possibility of scheduling a job onto a different resource type. However, these efforts [10], [21], [34], [37] consider only metrics like minimizing the makespan of the jobs being scheduled, and does not include notion of urgency or priority among the jobs.

Ahmad *et al.* [12] developed a scheduling scheme for jobs with fixed priority on heterogeneous computing systems, where they dynamically compute the precedence for jobs on different resources to make scheduling decisions. However, unlike our work, they consider scheduling for dependent set of tasks represented as directed acyclic graphs (DAG). In another work, scheduling of multiple DAGs on to heterogeneous systems [48] has been considered. Scheduling for jobs with hard deadlines [22] has also been considered in the context of heterogeneous multiprocessors. We believe that such a consideration is only realistic for real-time systems, and not for HPC jobs or systems. Our work differs from all the above works, in that, we use a value function which allows us to capture both *importance* and *urgency* of the jobs. Moreover, our work considers effective ways for sharing a single resource, which has not been considered previously.

There is also an extensive literature on schedulers that consider market-based or economy-based or user-centric approach for scheduling [11], [14], [15], [17], [26], [30], [33], [41], [42]. Millenium [17] is a user-centric scheduler based on the *FirstPrice* heuristic, which gives precedence for the job that gives the highest value per unit execution. Irwin *et al* [26] extended the work to consider *FirstReward*, which considers not only gains that can be obtained from scheduling the job, but also the cost involved in choosing one job over the other. Libra [42] is a budget-based proportional sharing scheduling scheme, where slack of the job and the amount of money a job is willing to pay is used. However, these efforts [17], [26], [42] consider only homogeneous resources. We have focused on heterogeneous resources, and our work has addressed several new challenges associated with heterogeneity for value-based scheduling.

VII. Conclusions

A wide adoption of heterogeneous computing has led to the evolution of programming models like OpenCL that offer portability across multi-core CPUs and GPUs. Batch schedulers for heterogeneous clusters that are in practice today neither exploit such portability nor pay adequate attention to user satisfaction or profitability in their scheduling considerations. In this paper, we have revisited the problem of scheduling independent jobs on heterogeneous computing clusters. We have proposed four novel scheduling schemes to dynamically map the jobs on to the heterogeneous resources. In addition, we also propose a novel heuristic that enables efficient resource sharing on multi-core CPUs and GPUs. We have extensively evaluated our schemes using our scheduler framework. Our comparisons against two baseline schemes, *TORQUE* and *MCT*, show that, a factor of up to 8.8 times yield improvement can be achieved. Finally, we show that by judiciously sharing the resources, we achieve an yield improvement of up to 23%.

References

[1] Amazon EC2. http://aws.amazon.com.
[2] Intel Mic Architecture. http://www.intel.com/content/www/us/en/architecture-and-technology/many-integrated-core/intel-many-integrated-core-architecture.html.
[3] Moab HPC Scheduler. http://www.adaptivecomputing.com/products/.
[4] OpenCL. http://www.khronos.org/opencl/.
[5] Parboil Benchmark Suite. http://impact.crhc.illinois.edu/parboil.php.
[6] PGI CUDA-X86 Compiler. http://www.pgroup.com/resources/cuda-x86.htm.
[7] RICC: Parallel Workload Logs. http://www.cs.huji.ac.il/labs/parallel/workload/l_ricc/index.html.
[8] Stampede Supercomputer. http://www.hpcwire.com/hpcwire/2011-09-22/.
[9] Top 500 Supercomputers. http://www.top500.org/.

[10] Torque Resource Manager. http://www.clusterresources.com/products/torque-resource-manager.php.

[11] D. Abramson, R. Buyya, and J. Giddy. A computational economy for grid computing and its implementation in the Nimrod-G resource broker. *Future Generation Comp. Syst.*, 18(8):1061–1074, 2002.

[12] I. Ahmad, M. Dhodhi, and R. Ul-Mustafa. DPS: Dynamic priority scheduling heuristic for heterogeneous computing systems. *IEE Proceedings - Computers and Digital Techniques*, 145(6):411–418, 1998.

[13] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier. StarPU: A unified platform for task scheduling on heterogeneous multicore architectures. *Concurrency and Computation: Practice and Experience, Special Issue: Euro-Par 2009*, 23:187–198, February 2011.

[14] R. Buyya, D. Abramson, and J. Giddy. An economy driven resource management architecture for global computational power grids. In *International Coference On Parallel and Distributed Processing Techniques and Applications*, 2000.

[15] R. Buyya, D. Abramson, and J. Giddy. An evaluation of economy-based resource trading and scheduling on computational power grids for parameter sweep applications. In *Sweep Applications, The Second Workshop on Active Middleware Services (AMS 2000)*, 2000.

[16] S. Che *et al*. Rodinia: A benchmark suite for heterogeneous computing. In *IISWC '09*, pages 44–54, Washington, DC, USA, 2009.

[17] B. N. Chun and D. E. Culler. User-centric performance analysis of market-based cluster batch schedulers. In *Proceedings of the 2nd IEEE/ACM International Symposium on Cluster Computing and the Grid*, pages 30–, Washington, DC, USA, 2002. IEEE Computer Society.

[18] G. F. Diamos and S. Yalamanchili. Harmony: An execution model and runtime for heterogeneous many core systems. In *HPDC '08*, pages 197–200, New York, NY, USA, 2008.

[19] G. F. Diamos *et al*. Ocelot: a dynamic optimization framework for bulk-synchronous applications in heterogeneous systems. In *PACT'10*, pages 353–364, New York, NY, USA, 2010.

[20] G. Fox. Editorial for economic models and algorithms for grid systems. *Concurrency and Computation: Practice and Experience*, 22(14):1971, 2010.

[21] R. F. t. Freund. Scheduling resources in multi-user, heterogeneous, computing environments with SmartNet. In *Proceedings of the Seventh Heterogeneous Computing Workshop*, HCW '98, pages 3–, Washington, DC, USA, 1998. IEEE Computer Society.

[22] S. Funk and S. K. Baruah. Restricting EDF migration on uniform heterogeneous multiprocessors. *Technique et Science Informatiques*, 24(8):917–938, 2005.

[23] K. Furlinger, N. J. Wright, and D. Skinner. Comprehensive performance monitoring for GPU cluster systems. In *Proceedings of the 2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and PhD Forum*, IPDPSW '11, pages 1377–1386, Washington, DC, USA, 2011. IEEE Computer Society.

[24] G. Giunta, R. Montella, G. Agrillo, and G. Coviello. A GPGPU transparent virtualization component for high performance computing clouds. In *Euro-Par 2010 - Parallel Processing*, volume 6271 of *Lecture Notes in Computer Science*, chapter 37, pages 379–391–391. Springer Berlin / Heidelberg, Berlin, Heidelberg, 2010.

[25] M. Harvey and G. D. Fabritiis. SWAN: A tool for porting CUDA programs to OpenCL. *Computer Physics Communications*, 182(4):1093 – 1099, 2011.

[26] D. E. Irwin, L. E. Grit, and J. S. Chase. Balancing risk and reward in a market-based task service. In *High performance Distributed Computing, 2004. Proceedings. 13th IEEE International Symposium on*, pages 160–169, 2004.

[27] A. Kerr, G. Diamos, and S. Yalamanchili. A characterization and analysis of PTX kernels. In *IISWC '09*, pages 3–12, Washington, DC, USA, 2009.

[28] A. Kerr, G. Diamos, and S. Yalamanchili. Modeling GPU-CPU workloads and systems. In *GPGPU '10*, pages 31–42, 2010.

[29] D. Kranzlmüller, U. Schwiegelshohn, Y. Robert, and F. F. Rivera. Topic 3: Scheduling and load balancing. In E. Luque, T. Margalef, and D. Benitez, editors, *Euro-Par*, volume 5168 of *Lecture Notes in Computer Science*, page 222. Springer, 2008.

[30] C. B. Lee and A. E. Snavely. Precise and realistic utility functions for user-centric performance analysis of schedulers. In *Proceedings of the 16th international symposium on High performance distributed computing*, HPDC '07, pages 107–116, New York, NY, USA, 2007. ACM.

[31] S. Lee and R. Eigenmann. OpenMPC: Extended OpenMP Programming and Tuning for GPUs. In *Proceedings of SC*, Nov 2010.

[32] C.-K. Luk, S. Hong, and H. Kim. Qilin: Exploiting parallelism on heterogeneous multiprocessors with adaptive mapping. In *MICRO '09*, pages 45–55, New York, NY, USA, 2009.

[33] M. Mahajan, A. Ramanathan, and M. Parashar. Active resource management for the differentiated services environment. *Int. Journal of Network Management*, 14(3):149–165, 2004.

[34] M. Maheswaran, S. Ali, H. J. Siegel, D. Hensgen, and R. F. Freund. Dynamic matching and scheduling of a class of independent tasks onto heterogeneous computing systems. In *Proceedings of the Eighth Heterogeneous Computing Workshop*, HCW '99, pages 30–, Washington, DC, USA, 1999. IEEE Computer Society.

[35] V. T. Ravi and G. Agrawal. A dynamic scheduling framework for emerging heterogeneous systems. *High-Performance Computing, International Conference on*, 0:1–10, 2011.

[36] V. T. Ravi, M. Becchi, G. Agrawal, and S. Chakradhar. Supporting GPU sharing in cloud environments with a transparent runtime consolidation framework. In *Proceedings of the 20th international symposium on High performance distributed computing*, HPDC '11, pages 217–228, New York, NY, USA, 2011. ACM.

[37] V. T. Ravi, M. Becchi, W. Jiang, G. Agrawal, and S. Chakradhar. Scheduling concurrent applications on a cluster of CPU-GPU nodes. In *Proceedings of the 2012 12th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*, CCGRID '12 (to appear), 2012.

[38] V. T. Ravi, W. Ma, D. Chiu, and G. Agrawal. Compiler and runtime support for enabling generalized reduction computations on heterogeneous parallel configurations. In *ICS '10*, pages 137–146, New York, NY, USA, 2010.

[39] G. Sabin, R. Kettimuthu, A. Rajan, and P. Sadayappan. Scheduling of parallel jobs in a heterogeneous multi-site environment. In *in Proc. of the 9th International Workshop on Job Scheduling Strategies for Parallel Processing*, pages 87–104, 2003.

[40] G. Sabin, V. Sahasrabudhe, and P. Sadayappan. Assessment and enhancement of meta-schedulers for multi-site job sharing. In *HPDC '05*, pages 144–153, Washington, DC, USA, 2005.

[41] E. Santos-Neto and K. Keahey. Poster reception - To bid or not to bid: a hybrid market-based resource allocation framework. In *SC*, page 167. ACM Press, 2006.

[42] J. Sherwani, N. Ali, N. Lotia, Z. Hayat, and R. Buyya. Libra: A computational economy-based job scheduling system for clusters. *Softw. Pract. Exper.*, 34:573–590, may 2004.

[43] J. Stratton, S. Stone, and W. mei Hwu. MCUDA: An Efficient Implementation of CUDA Kernels for Multi-Core CPUs. In *21st Annual Workshop on Languages and Compilers for Parallel Computing (LCPC'2008)*, July 2008.

[44] D. Talby and D. G. Feitelson. Improving and stabilizing parallel computer performance using adaptive backfilling. In *IPDPS*, 2005.

[45] G. Teodoro *et al*. Coordinating the use of GPU and CPU for improving performance of compute intensive applications. In *CLUSTER '09*, pages 1–10, 2009.

[46] D. Tsafrir, Y. Etsion, and D. G. Feitelson. Backfilling using system-generated predictions rather than user runtime estimates. *IEEE Trans. Parallel Distrib. Syst.*, 18(6):789–803, 2007.

[47] J. B. Weissman and P. Srinivasan. Ensemble scheduling: Resource co-allocation on the computational grid. In C. A. Lee, editor, *GRID*, volume 2242 of *Lecture Notes in Computer Science*, pages 87–98. Springer, 2001.

[48] H. Zhao and R. Sakellariou. Scheduling multiple dags onto heterogeneous systems. In *IPDPS*. IEEE, 2006.